Foundations of Coordination and Contracts and Their Contribution to Session Type Theory

Mario Bravetti, Gianluigi Zavattaro

Foundations of Coordination and Contracts and their Contribution to Session Type Theory

Mario Bravetti Gianluigi Zavattaro

Department of Computer Science and Engineering & Focus Team, INRIA
University of Bologna, Italy

Abstract. We briefly recall results obtained in twenty years of research, spanning across the old and the new millennium, on the expressiveness of coordination languages and on behavioural contracts for Service-Oriented Computing. Then, we show how the techniques developed in those contexts are currently contributing to the clarification of aspects that were unclear about session types, in particular, asynchronous session subtyping that was considered decidable since 2009, while it was proved to be undecidable in 2017.

1 Introduction

Shared dataspaces and the so-called generative communication paradigm [27] attracted a lot of attention since the initial years of research about foundations of coordination models and languages. Linda [18], probably the most popular language based on this coordination model, is based on the idea that concurrent processes interact via a shared dataspace, the so-called Tuple Space (TS for short), where the information needed to coordinate the activities are introduced and retrieved. After its insertion in the TS, a datum becomes equally accessible to all processes, but it is bound to none. In this way, the interaction among concurrent processes is decoupled in space and time, principles useful in the development of modular and scalable concurrent and distributed systems.

Concerning foundational studies on Linda-like coordination languages, it appeared immediately clear that techniques borrowed from the tradition of concurrency theory could be naturally applied. At the first two editions of the Coordination conference, two process calculi based on Linda were proposed by De Nicola and Pugliese [22] and by Busi, Gorrieri and Zavattaro [14]. In particular, the latter started a line of research on the expressiveness of Linda-like coordination primitives that exploited, besides process calculi, also Petri nets. For instance, Petri nets were used in [13], to prove that a Linda process calculus with input, output and test-for-absence is *not* Turing complete if the semantics for output is *unordered*, i.e., there is an unpredictable delay between the execution of an output and the actual availability of the emitted datum in the TS. It is interesting to recall that, on the other hand, the same calculus is Turing complete if an *ordered* semantics is considered, i.e. an emitted datum is immediately available in the TS after the corresponding output is executed. Turing completeness was proved by showing an encoding of a Turing powerful formalism, namely

Random Access Machines (RAMs), which is a computational model based on registers that can be incremented, decremented and tested for emptiness.

The success of the Linda coordination model was witnessed by the development, at the end of the 90s, of Linda-based middlewares from main ICT vendors like IBM and Sun Microsystem, which proposed T-Spaces and JavaSpaces, respectively. The basic Linda coordination model was extended with primitives for event notification, time-out based data expiration, and transactions. The techniques for evaluating the expressive power of Linda languages had to become more sophisticated to cope with these additional primitives. In particular, Petri nets with transfer and reset arcs [23] were adopted to cope also with the new coordination mechanisms for event notification [16] and for temporary data [15].

During the initial years of the new millennium, Service-Oriented Computing (SOC) emerged as an alternative model for developing communication-based distributed systems. In particular, the large diffusion of Web Services called for the development of new languages and techniques for service composition. The idea, at the basis of SOC, is to conceive an ecosystem of services that expose operations that can be combined to realize new applications. To support this idea, it is necessary for the services to be equipped with an interface that, besides describing the offered operations and the format of the exchanged messages, defines the conversation protocols, i.e., the expected flow of invocations of the operations. These interfaces, in particular the specification of the conversation protocol, are also called *behavioural contracts*.

Process calculi contributed to the development of theories for behavioural contracts. This line of research was initiated by Carpineti, Castagna, Laneve and Padovani [17], for the case of client-server composition, and by Bravetti and Zavattaro [6], for multiparty service compositions. The latter is particularly significant for the so-called service choreographies, i.e., systems in which there exists no central orchestrator, responsible for invoking all the other services in the system, because services reciprocally interact. Behavioural contract theories focused mainly on the investigation of appropriate notions of correctness for service compositions (i.e., define when a system based on services is free from communication errors) and on the characterization of notions of compatibility between services and behavioural contracts (i.e., define when a service conforms to a given behavioural contract).

For multiparty composition, a fairness based notion of correctness, called *compliance*, was proposed for the first time in [6]: a system is correct if, whatever state can be reached, there exists a continuation of the computation that yields a final state in which *all* services have successfully completed. Given the notion of compliance, it is possible to define also a natural notion of *refinement* for behavioural contracts: a refinement is a relation among contracts such that, given a set of compliant contracts $C_1, \cdots, C_n$, each contract C_i can be *independently* replaced by any of its possible refinements C'_i, and the overall system obtained by composition of $C'_1, \cdots, C'_n$ is still compliant. Contract refinement can then be used to check whether a service conforms with a behavioural contract: it is sufficient to verify if the communication behaviour of the service refines the

behavioural contract. This, in fact, implies that such service can be safely used wherever a service is expected with the behaviour specified by the contract.

A negative result in the theory of behavioural contracts is that, in general, the union of two refinement relations is not guaranteed to be itself a refinement. This implies the impossibility to define a maximal notion of refinement. For this reason, most of the effort in the line of research on behavioural contracts initiated in [6] has been dedicated to the identification of interesting subclasses of contracts for which the maximal refinement exists. Such classes are: contracts with *output persistence* [6] (i.e. output actions cannot be avoided when a state is entered in which they are ready to be executed), contract refinement preserving *strong compliance* [7] (i.e. as soon as an output is ready to be executed, a receiver is guaranteed to be ready to receive it), and *asynchronously communicating* contracts [10] (i.e. communication is mediated by fifo buffers). In the first two of these three cases, it has been also possible to provide a sound algorithmic characterization of the corresponding maximal refinements.

To the best of our knowledge, characterizing algorithmically the maximal contract refinement in case of asynchronous communication is still an open problem. The main source of difficulty derives from the fact that, due to the presence of unbounded communication buffers, systems of asynchronously communicating contracts are infinite-state, even if contracts are finite-state. In the light of this difficulty, we tried to take inspiration from work on session types, where asynchronous communication has been investigated since the seminal work by Honda, Yoshida and Carbone [29] (recipient of the most influential POPL'08 paper award). Session types can be seen as a simplification of contracts obtained by imposing some limitations: there are only two possible choices, *internal* choice among distinct outputs and *external* choice among distinct inputs.

The counterpart of contract refinement in the context of session types is *subtyping* [26]. If we consider asynchronous communication, both contract refinement and session subtyping can admit a refinement/subtype to perform the communication actions in a different order. For instance, given a contract/type that performs an input followed by an output, a refinement/subtype can *anticipate* the output before the input, because such output can be *buffered* and actually received afterwards. Asynchronous session subtyping was already studied by Mostrous, Yoshida, and Honda in [38], where also an algorithm for checking subtyping was presented. Upon studying this algorithm we noticed an error in its proof of termination: if, while checking subtyping, the buffer grows unboundedly, the proposed procedure does not terminate. Subsequently, Bravetti, Carbone and Zavattaro [4] and Lange and Yoshida [33] independently proved that asynchronous session subtyping is actually undecidable. Our experience in the modeling of Turing complete formalism (see the above discussion about encoding RAMs in the Linda process calculus) helped in finding an appropriate Turing powerful model to be encoded in terms of asynchronous session subtyping. In particular, we were able to present a translation from a Queue Machine M (a computational model similar to pushdown automata, but with a queue instead of a stack) to a pair of session types that are in asynchronous subtyping relation

if and only if M does not terminate. Then, undecidability of asynchronous session subtyping directly follows from the undecidability of the halting problem for Queue Machines.

These negative results opened the problem of identifying significant classes of session types for which asynchronous subtyping can be decided. Currently, the most interesting fragments have been identified in [4,33] and [5]. In the former, an algorithm is presented for the case in which one of the two types is completely deterministic, i.e. all choices –both internal and external– have one branch only. In the latter, we have considered single-out (and single-in) session types, meaning that in both types to be checked all internal choices (resp. external choices) have one branch only. In the design of our algorithm we have been inspired by our expertise in the analysis of the expressiveness of Linda process calculi. In particular, the analysis techniques in Petri nets with transfer and reset arcs (our tools to prove decidability results) are based on the notion of well quasi ordering (wqo): while generating an infinite sequence of elements, it is guaranteed to eventually generate an element that is greater –with respect to the wqo– of an already generated element. Similarly to the procedure in [38], our algorithm checks a sequence of judgements, but differently from [38], termination is guaranteed because there exists a wqo on judgements, and the algorithm can terminate when a judgement greater than an already checked one is considered.

Outline of the paper. The remainder of this paper is divided in three main parts: in Section 2 we discuss the techniques used to investigate the expressive power of the Linda coordination model; in Section 3 we recall the main results concerning behavioural contracts; and in Section 4 we present recent results on session types for asynchronous communication. The additional Section 5 reports some concluding remarks.

2 Process Calculi to Study the Expressiveness of Linda

Several Linda process calculi have been proposed in the literature, like PAL [22], LLinda [21], and the Linda calculi in [14] and [12], just to mention some of them (published in the proceedings of the first two editions of the Coordination conference). Those calculi have been defined for several purposes: investigate from a foundational viewpoint coordination models and languages, develop novel formal analysis techniques for coordinated systems, or drive the implementation of fully-fledged coordination languages. In this section, we focus on process calculi used to prove results about the expressive power of primitives for Tuple Spaces. Different techniques have been adopted, spanning from Turing completeness (used, e.g., in [13]) to modular embeddings (used, e.g., in [12]). To give the reader a more precise idea of such techniques, in particular the former, we report some results taken from [13,16].

We start by introducing a Linda calculus with four basic primitives: (i) *out* to introduce a new datum into a shared repository called dataspace, (ii) *in* to consume one datum available in the repository, (iii) *notify* to register the interest

$$\texttt{PRE}: \ \frac{j \in I}{\sum_{i \in I} \alpha_i.P_i \xrightarrow{\alpha_j} P_j} \qquad \texttt{REPL}: \ !\alpha.P \xrightarrow{\alpha} !\alpha.P \mid P \qquad \texttt{PAR}: \ \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$$

Table 1. The transition system for processes (symmetric rule of PAR omitted).

in future emissions of a specific datum (when such datum will be emitted a new instance of a given process will be spawned), and (iv) *tfa* to test for the absence of a specific datum. This calculus is Turing complete (result taken from [13]) while the fragment without *tfa* is not (result taken from [16]). The proof of Turing completeness is by reduction from Random Access Machines, while the non Turing completeness of the considered fragment was proved in [16] by resorting to a (non Turing complete) variant of Petri net, namely Petri nets with transfer arcs (see, e.g., [23]); here, we present an alternative proof technique based on well quasi orderings (which are actually used to prove decidability results for Petri nets with transfer arcs).

We now formally report the definition of a Linda-based process calculus, starting from the syntax of processes.

Definition 1 (Linda Processes). *Let Name, ranged over by a, b, ..., be a denumerable set of names. Processes are defined by the following grammar:*

$$\begin{array}{lcl} \alpha & ::= & \mathit{in}(a) \ \mid \ \mathit{out}(a) \ \mid \ \mathit{notify}(a, P) \ \mid \ \mathit{tfa}(a) \\ P & ::= & \sum_{i \in I} \alpha_i.P_i \ \mid \ !\alpha.P \ \mid \ P|P \end{array}$$

The basic process actions are $\mathit{in}(a)$ and $\mathit{out}(a)$ denoting the consumption or emission, respectively, of one instance of datum a from/into the shared dataspace. Two additional primitives are considered: $\mathit{notify}(a, P)$ to register a listener interested in future emissions of the datum a (the reaction to such event will be the spawning of process P) and $\mathit{tfa}(a)$ to test for the absence of the datum a. The term $\sum_{i \in I} \alpha_i.P_i$ denotes a process ready to perform any of the action α_i, and then proceed by executing the corresponding continuation P_i. We use $\mathbf{0}$ to denote such process in case $I = \emptyset$, and we will usually omit trailing $\mathbf{0}$. The replicated process $!\alpha.P$ performs an initial action α and then spawns the continuation P by keeping $!\alpha.P$ in parallel. Two parallel processes P and Q are denoted with $P|Q$. In the following we will use the notation $\prod_{i \in I} P_i$ to denote the parallel composition of processes indexed on the set of indexes I.

We now formalize the operational semantics for Linda processes in terms of a transition system with four kinds of labels: $\mathit{in}(a)$, $\mathit{out}(a)$, $\mathit{notify}(a, P)$, and $\mathit{tfa}(a)$. The transition system is the least one satisfying the axioms and rules reported in Table 1. The PRE rule simply allows a sum process to execute one of its initial actions and then continue with the corresponding continuation. REPL allows $!\alpha.P$ to execute α, spawn an instance of the continuation P, and keep $!\alpha.P$ in parallel. Finally, PAR allows a parallel process to execute an action.

$$\frac{P \xrightarrow{in(a)} P'}{\langle P, \mathcal{S} \uplus a, \mathcal{L}\rangle \to \langle P', \mathcal{S}, \mathcal{L}\rangle} \qquad \frac{P \xrightarrow{notify(a,Q)} P'}{\langle P, \mathcal{S}, \mathcal{L}\rangle \to \langle P', \mathcal{S}, \mathcal{L} \uplus (a, Q)\rangle}$$

$$\frac{P \xrightarrow{out(a)} P' \quad \mathcal{L} = \{(a, P_i) | i \in I\} \uplus \{(b_j, Q_j) \mid \forall j. b_j \neq a\}}{\langle P, \mathcal{S}, \mathcal{L}\rangle \to \langle P' | \prod_{i\in I} P_i, \mathcal{S} \uplus a, \mathcal{L}\rangle}$$

$$\frac{P \xrightarrow{tfa(a)} P' \quad a \notin \mathcal{S}}{\langle P, \mathcal{S}, \mathcal{L}\rangle \to \langle P', \mathcal{S}, \mathcal{L}\rangle}$$

Table 2. The reduction relation for systems (brackets in singletons are omitted).

We now move to the syntax and semantics of systems, in which processes are equipped with a dataspace and a multiset of registered listeners.

Definition 2 (Linda Systems). *A system S is a tuple $\langle P, \mathcal{S}, \mathcal{L}\rangle$ where P is a process, $\mathcal{S}$ is the dataspace (i.e. a multiset over Name), and $\mathcal{L}$ are the registered listeners (i.e. a multiset of pairs (a, P)).*

The semantics of systems is defined by the minimal transition system satisfying the rules in Table 2. The transitions for systems allow processes to (i) consume data from the shared dataspace, (ii) register a new listener, (iii) introduce a new datum in the shared dataspace with the corresponding spawning of the processes in the interested listeners, and (iv) test for the absence of one datum in the dataspace. We use $\uplus$ to denote multiset union.

In the following, we will consider *termination* and *divergence* of systems.

Definition 3 (Termination and Divergence). *Given a system S, we say that S terminates, denoted $S\downarrow$, if there exists at least an ending computation, i.e., there exist $S_1, \cdots, S_n$ such that $S \to S_1 \to S_2 \cdots \to S_n \not\to$, where $S_n \not\to$ means that there exists no system S' s.t. $S_n \to S'$. We say that S diverges, denoted $S\uparrow$, if there exists at least one infinite computation, i.e., there exist one infinite sequence of systems $S_1, \cdots, S_n, \cdots$ such that $S \to S_1 \to S_2 \cdots \to S_n \to \cdots$.*

2.1 Turing Completeness

The Turing completeness of the Linda calculus was proved in [13]. Actually, the calculus in that paper considered a variant of the *tfa* primitive called *inp*, which is a non-blocking version of *in*: *inp* has two possible continuations P and Q, the first one activated in case the consumption succeeds, and a second one activated if the message of interest is absent. This primitive coincides with the process $in(a).P + tfa(a).Q$. The proof of Turing completeness was based on an encoding of Random Access Machines (RAMs) [42], a well known register-based Turing powerful formalism. Here, we rephrase that encoding by exploiting *in*, *out* and *tfa*, without using *notify*. For this reason, in this subsection, we do not consider listeners and denote systems simply with pairs $\langle P, \mathcal{S}\rangle$ of processes and dataspace.

A RAM, denoted in the following with R, is a computational model composed of a finite set of registers $r_1, \ldots, r_n$, that can hold arbitrarily large natural numbers, and of a program composed by indexed instructions $(1 : I_1), \ldots, (m : I_m)$, that is a sequence of numbered instructions, like arithmetic operations (on the contents of registers) or conditional jumps. An internal state of a RAM is given by $(i, c_1, \ldots, c_n)$ where i is the program counter indicating the next instruction to be executed, and $c_1, \ldots, c_n$ are the current contents of the registers $r_1, \ldots, r_n$, respectively.

The computation starts from the first instruction $(1 : I_1)$ and terminates when the program counter points to an undefined instruction. In other terms, the initial configuration is $(1, 0, \ldots, 0)$ and the computation continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction outside the valid range $1, \ldots, m$ is reached.

Formally, we indicate by $(i, c_1, \ldots, c_n) \rightarrow_R (i', c'_1, \ldots, c'_n)$ the fact that the configuration of the RAM R changes from $(i, c_1, \ldots, c_n)$ to $(i', c'_1, \ldots, c'_n)$ after the execution of the i-th instruction.

In [36] it is shown that the following two instructions are sufficient to model every recursive function: $(i : Succ(r_j))$ to add 1 to the content of register r_j; $(i : DecJump(r_j, s))$ that, if the content of register r_j is not zero, decreases it by 1 and goes to the next instruction, otherwise jumps to instruction s.

We start presenting how to encode RAM instructions into Linda processes:

$$\begin{aligned} [\![(i : Succ(r_j))]\!] &= !in(p_i).out(r_j).out(p_{i+1}) \\ [\![(i : DecJump(r_j, s))]\!] &= !in(p_i).\big(\ in(r_j).out(p_{i+1}) \ + \ tfa(r_j).out(p_s)\ \big) \end{aligned}$$

The idea is to represent the content of the register r_j with a corresponding number of instances of the datum r_j in the dataspace. The program counter is modeled by a datum p_i indicating that the i-th instruction is the next one to be executed. The modeling of the i-th instruction always starts with the consumption of the p_i datum. An increment instruction on r_j simply produces one datum r_j, while a $DecJump$ instruction either consumes one datum r_j or tests for the absence of such datum. After these operations, the subsequent program counter datum is emitted.

We now present the full definition of our encoding. Let R be a RAM with m instructions, and let $(i, c_1, \ldots, c_n)$ be one of its configurations. With

$$[\![(i, c_1, \ldots, c_n)]\!]_R = \langle \prod_{1 \leq i \leq m} [\![(i : I_i)]\!], \{p_i, \underbrace{r_1, \cdots, r_1}_{c_1 \ times}, \cdots, \underbrace{r_n, \cdots, r_n}_{c_n \ times}\} \rangle$$

we denote the system representing the configuration $(i, c_1, \ldots, c_n)$.

We now formally recall the correctness of the encoding (proved in [13]) from which we conclude the Turing completeness of the Linda calculus.

Theorem 1. *Let R be a RAM. Given $\rightarrow^3$ as a notation for three successive reductions of systems, we have that:*

- *Soundness: if $[\![(i, c_1, \ldots, c_n)]\!]_R \rightarrow^3 Q$ then there exists a unique configuration $(j, c'_1, \ldots, c'_n)$ such that $Q = [\![(j, c'_1, \ldots, c'_n)]\!]_R$ and $(i, c_1, \ldots, c_n) \rightarrow_R (j, c'_1, \ldots, c'_n)$*

– *Completeness: if* $(i, c_1, \ldots, c_n) \rightarrow_R (j, c'_1, \ldots, c'_n)$ *then* $[\![(i, c_1, \ldots, c_n)]\!] \rightarrow^3 [\![(j, c'_1, \ldots, c'_n)]\!]$

As a consequence of the Turing completeness of the Linda calculus, we have that both termination and divergence are undecidable, namely, given a system S, it is in general undecidable whether $S\downarrow$ or $S\uparrow$. Notice that Turing completeness does not depend on *notify*, as this primitive is not used in the modeling of RAMs reported in the previous subsection.

2.2 Decidability of Divergence in the Calculus without *tfa*

We now focus on the expressive power of *tfa*, by investigating whether the calculus continues to be Turing complete even if we remove such primitive. Hence, we consider the fragment of the Linda calculus without the *tfa* primitive. We will observe that this fragment is no longer Turing powerful, as divergence turns out to be decidable, namely, given a system S, it is always possible to decide whether $S\uparrow$. This result was proved in [16] by presenting an encoding of a Linda calculus with *in*, *out*, and *notify* into Petri nets with transfer arcs: an extension of place/transition Petri nets for which properties like coverability, or the existence of an infinite firing sequence, are still decidable. In this section we present a novel alternative proof (at least for this considered Linda calculus) inspired by the theory of well structured transition systems (WSTS) [24]: we first define an ordering on systems configurations which is proved to be a well quasi ordering, and then we show that the operational semantics is compatible with such ordering.

We start by recalling the notion of well quasi ordering (see, e.g., [24]).

Definition 4. *A reflexive and transitive relation on a set X is called* quasi ordering. *A* well quasi ordering *(wqo) is a quasi ordering $(X, \leq)$ such that, for every infinite sequence $x_1, x_2, \cdots$, there exist $i < j$ with $x_i \leq x_j$.*

In the following we will use the following well known results for wqo:

– Consider a finite set S and the set of its multisets $\mathcal{M}(S)$. We have that multiset inclusion is a well quasi ordering for the latter, namely $(\mathcal{M}(S), \subseteq)$ is a wqo, where $\subseteq$ denotes multiset inclusion.
– Consider k well quasi orderings $(X_1, \leq_1), \cdots, (X_k, \leq_k)$. Let Π be the cartesian product $X_1 \times \cdots \times X_k$ and $\leq^k$ be the natural extension of the orderings $\leq_1, \cdots, \leq_k$ to Π, i.e., $(x_1, \cdots, x_k) \leq^k (y_1, \cdots, y_k)$ if and only if $x_1 \leq_1 y_1, \cdots, x_k \leq_k y_k$. We have that $(\Pi, \leq^k)$ is a wqo.

We now recall, taking it from [24], the notion of compatibility[1] of a transition system w.r.t. an ordering.

Definition 5. *A transition system $(X, \rightarrow)$ is* compatible *with respect to an ordering $(X, <)$ if, given two states $s, t \in X$ of the transition system such that $s < t$ and $s \rightarrow s'$ for some s', then there exists t' such that $s' < t'$ and $t \rightarrow t'$.*

[1] The compatibility notion used in this paper is named *strict* compatibility in [24].

A known result from the theory of WSTS (Theorem 4.6 in [24]) is that it is decidable to establish the existence of an infinite computation in a transition system compatible with a wqo. To exploit this result, we now define a wqo on systems $\langle P, \mathcal{S}, \mathcal{L}\rangle$. To do this we will make use of the above result stating that multiset inclusion over multisets with a finite domain is a wqo. This result can be directly applied to dataspaces $\mathcal{S}$ and registered listeners $\mathcal{L}$ as they are multisets, while this is not possible on processes P that are terms with a given syntax. Hence, it is necessary to give an interpretation of such terms P as multisets: this can be obtained by extracting from a term P the multiset of sequential or replicated processes constituting P. Formally, given the process P we define inductively, on its syntactic structure, the multiset $m(P)$ as follows:

$$m(\sum_{i\in I}\alpha_i.P_i) = \{\sum_{i\in I}\alpha_i.P_i\} \qquad m(!\alpha.P) = \{!\alpha.P\} \qquad m(P|Q) = m(P) \uplus m(Q)$$

We are now ready to define the following ordering on systems:

$$\langle P, \mathcal{S}, \mathcal{L}\rangle \leq_S \langle P', \mathcal{S}', \mathcal{L}'\rangle \quad \Leftrightarrow \quad m(P) \subseteq m(P') \wedge \mathcal{S} \subseteq \mathcal{S}' \wedge \mathcal{L} \subseteq \mathcal{L}'$$

We now prove that this ordering $\leq_S$ on systems is a wqo for the set of systems that are reachable from a given initial system.

Proposition 1. *Let $S_0 = \langle P_0, \mathcal{S}_0, \mathcal{L}_0\rangle$ be an initial system and let Sys be the set of systems that are reachable from S_0 according to the reduction relation defined in Table 2. We have that $(Sys, \leq_S)$ is a wqo.*

Proof. Let $S = \langle P, \mathcal{S}, \mathcal{L}\rangle$ be a system reachable from the initial system $S_0 = \langle P_0, \mathcal{S}_0, \mathcal{L}_0\rangle$. It is easy to see that the data in $\mathcal{S}$ and the listeners in $\mathcal{L}$ are taken from finite domains given, respectively, by the parameters of the primitives *out* and *notify* occurring in the initial process P_0 (plus data or listeners already available in $\mathcal{S}_0$ or $\mathcal{L}_0$). We now observe that also the sequential or replicated processes in $m(P)$ are taken from a finite domain. In fact, P is the parallel composition of terms that already occur in the initial process P or in the listeners in $\mathcal{L}_0$. This is because the reduction relation defined in Table 2 does not generate new processes, but simply consumes initial actions in front of already available sequential or replicated processes or spawns processes already present in listeners.

Hence (for the first of the two well known results recalled for wqo) we can conclude that multiset inclusion is a wqo for the multisets $m(P)$ associated to the reachable processes P, as well as for the reachable dataspaces $\mathcal{S}$ and listeners $\mathcal{L}$. The ordering $\leq_S$ is the natural ordering for the cartesian product of these last three wqo, hence it is also a wqo (for the second of the recalled results). □

We now observe that the reduction relation for systems defined in Table 2 is compatible with the ordering $\leq_S$.

Proposition 2. *Let S, S' and T be three systems, in which the tfa primitive does not occur, such that $S \to S'$ and $S \leq_S T$. We have that there exists a system T' such that $T \to T'$ and $S' \leq_S T'$.*

Proof. We first observe that, for every pair of processes P and P' such that $m(P) \subseteq m(P')$, if $P \xrightarrow{\alpha} Q$ then there exists also Q' such that $P' \xrightarrow{\alpha} Q'$ and $m(Q) \subseteq m(Q')$. This can be proved by induction on the structure of P. If $P = \sum_{i \in I} \alpha_i.R_i$ (resp. $P = !\alpha.R$) then P' is the parallel composition of terms including also P. Let Q be the process in the r.h.s. of the transition inferred on P by the rule `PRE` (resp. `REPL`) in Table 1. Such transition can be inferred (by the same rule plus possible successive applications of `PAR`) also on P': let Q' be the process in the r.h.s. of such transition. As the same `PRE` (resp. `REPL`) rule is initially applied, we have that Q' is the parallel composition of processes including also Q, hence $m(Q) \subseteq m(Q')$. In the inductive case, we have that the last rule applied in the transition $P \xrightarrow{\alpha} Q$ is `PAR`, and the thesis directly follows from the inductive hypothesis on the transition used in the premise of the application of `PAR`.

Consider now three systems S, S' and T, in which the *tfa* primitive does not occur, such that $S \leq_S T$ and $S \to S'$. The latter implies that, given $S = \langle P, \mathcal{S}_S, \mathcal{L}_S \rangle$ and $S' = \langle P', \mathcal{S}_{S'}, \mathcal{L}_{S'} \rangle$, there exists α such that $P \xrightarrow{\alpha} P'$. Consider now $T = \langle Q, \mathcal{S}_T, \mathcal{L}_T \rangle$. Having $S \leq_S T$, we also have $m(P) \subseteq m(Q)$. For the above observation, then also $Q \xrightarrow{\alpha} Q'$ for a process Q' such that $m(P') \subseteq m(Q')$. Given this transition $Q \xrightarrow{\alpha} Q'$, by the rules in Table 1, we also have that $T \to T' = \langle Q', \mathcal{S}_{T'}, \mathcal{L}_{T'} \rangle$ with $S' \leq_S T'$. The latter is a consequence of $m(P') \subseteq m(Q')$, $\mathcal{S}_{S'} \subseteq \mathcal{S}_{T'}$ and $\mathcal{L}_{S'} \subseteq \mathcal{L}_{T'}$. The last two statements follow from the fact that $\mathcal{S}_{T'}$ (resp. $\mathcal{L}_{T'}$) is obtained from $\mathcal{S}_T$ (resp. $\mathcal{L}_T$) by applying the same modification applied to $\mathcal{S}_S$ (resp. $\mathcal{L}_S$) to obtain $\mathcal{S}_{S'}$ (resp. $\mathcal{L}_{S'}$); hence multiset inclusion $\mathcal{S}_S \subseteq \mathcal{S}_T$ (resp. $\mathcal{L}_S \subseteq \mathcal{L}_T$) is preserved. □

We can finally conclude with our decidability result.

Theorem 2. *Let $S = \langle P, \mathcal{S}, \mathcal{L} \rangle$ be a system in which the tfa primitive does not occur. It is decidable whether $S\uparrow$.*

Proof. Direct consequence of Proposition 1, Proposition 2 and the result taken from the theory of WSTS (Theorem 4.6 in [24]) recalled above. □

We conclude by recalling other related results about the un/decidability of $S\downarrow$, i.e. the existence of a terminating computation. In [16] it is proved that $S\downarrow$ is undecidable in the fragment of the Linda calculus, without *tfa*, considered in this subsection: hence we conclude that the Linda calculus with *in*, *out* and *notify* is in between decidability and undecidability, namely, $S \uparrow$ is decidable while $S\downarrow$ is not. The undecidability proof consists of an encoding of RAMs which is nondeterministic: the Linda system corresponding to a RAM could have several alternative computations, but all the computations that are not faithful w.r.t. the modeled RAM are guaranteed to be divergent. Hence, the Linda system has a terminating computation if and only if the corresponding RAM terminates.

In the same paper [16], it is also discussed that if we remove also the primitive *notify* from the calculus, also $S \downarrow$ becomes decidable. This follows from the possibility to faithfully encode the fragment of the calculus with only *in* and *out* into classical place/transition Petri nets, for which it is possible to decide the

reachability of any marking from which no other transitions can be fired. This additional result allows us to conclude that adding the primitive *notify* strictly increases the expressive power of the Linda calculus with only *in* and *out*.

3 Behavioural Contracts

Behavioural contracts are used to describe the message-passing behaviour of processes. The adoption of process calculi to the specification and analysis of behavioural contracts was initiated by Fournet et al. [25], who proposed to specify contracts as CCS-like processes. They also defined a notion of conformance between processes and contracts following a substitution principle: a process conforms to a contract if it can replace it in any context without adding additional stuck behaviour. Contract have been subsequently studied in the context of service oriented computing: contracts for client-service interaction have been proposed by Carpineti et al. [17] and then independently extended along different directions by, e.g., Bravetti and Zavattaro (see e.g. [6,7,8]) by Laneve and Padovani [32], by Castagna et al. [19], and Barbanera and de'Liguoro [1].

All such theories of contracts introduce, under different assumptions, notions of *contract refinements* that can be seen as generalizations of the notion of conformance initially studied in [25]: a contract refines another one if it can safely replace it in any possible context. To give to the reader an idea of such techniques, here we report a contract theory discussed in [8,9], for synchronous communication, and [10], for the asynchronous case. In particular, the latter represents the unique contract theory, to the best of our knowledge, specifically tailored to asynchronous communication.

More precisely, the contract theory that we present is based on the following ingredients: the notion of correct contract composition, the definition of contract refinement, and its algorithmic characterization. Both synchronous and asynchronous communication, excluding the algorithmic characterization which is available only for the synchronous case.

We start by presenting the formal definition of behavioural contracts as it appears in [2]. Contracts can be seen as a representation of the communication actions that can be performed at a certain location over the network. We assume a denumerable set of action names $\mathcal{N}$, ranged over by $a, b, c, \ldots$ and a denumerable set Loc of location names, ranged over by $l, l', l_1, \cdots$. We use $\tau \notin \mathcal{N}$ to denote an internal (unsynchronizable) action. Contracts are denoted adopting a basic process algebra with prefixes over $\{\tau, a, \overline{a}_l \mid a \in \mathcal{N}, l \in Loc\}$, denoting internal, input, and output action, respectively. Notice that a destination location is specified for outputs. Such a process algebra is a simple extension of basic CCS [35] with successful termination denoted by "$\mathbf{1}$" (whereas the traditional null process "$\mathbf{0}$" denotes a failure or a deadlock).

Definition 6 (Behavioural Contracts). *We consider a denumerable set of contract variables Var ranged over by X, Y, $\cdots$. The syntax of contracts is*

$$\mathbf{1} \xrightarrow{\surd} \mathbf{0} \qquad\qquad \alpha.C \xrightarrow{\alpha} C$$

$$\frac{C \xrightarrow{\lambda} C'}{C+D \xrightarrow{\lambda} C'} \qquad\qquad \frac{C\{recX.C/X\} \xrightarrow{\lambda} C'}{recX.C \xrightarrow{\lambda} C'}$$

Table 3. Semantic rules for contracts (symmetric rules omitted).

defined by the following grammar

$$\begin{array}{lcl} C & ::= & \mathbf{0} \mid \mathbf{1} \mid \alpha.C \mid C+C \mid X \mid recX.C \\ \alpha & ::= & \tau \mid a \mid \overline{a}_l \end{array}$$

where $recX.\_$ is a binder for the process variable X denoting recursive definition of processes. We assume that in a contract C all process variables are bound. In the following we will omit trailing "$\mathbf{1}$" when writing contracts.

The operational semantics of contracts is defined in terms of a transition system labeled over $L = \{a, \overline{a}_l, \tau, \surd \mid a \in \mathcal{N}, l \in Loc\}$, ranged over by λ, λ', ..., obtained by the rules in Table 3 (plus a symmetric rule for choice). We use the notation $C\{\_/\_\}$ to denote syntactic replacement. Semantic rules are the standard ones, apart from that of term $\mathbf{1}$, which performs a $\surd$ transition denoting successful termination. The semantics of a contract C yields a *finite-state* labeled transition system,[2] whose states are the contracts reachable from C.

We now present a simple example of a contract describing an authentication service that repeatedly performs two kinds of task: (*i*) the authentication of clients by receiving their username and password, and (*ii*) the request to an external account service for update of the list of the registered users.

$$recX.(\ username.password.(\overline{accepted}_{client}.X + \overline{failed}_{client}.X) + \overline{updateAccounts}_{accountServer}.newAccounts.X\ )$$

The contract indicates a repeated choice between the two possible tasks. The first task is activated by the reception of an invocation on *username*. In this case, a *password* should subsequently be received and then two possible answers are sent back to the *client*: either *accepted* or *failed*. The second task is activated by sending a request for update to the *accountServer*. In this case, the *newAccounts* are subsequently received.

In the following we will study independent contract refinement. As already anticipated in the Introduction under *synchronous* communication a maximal independent contract refinement that preserves compliance does not exist. In [6] we showed that this is a consequence of the symmetry between input and output actions and that a possible solution, for synchronous communication, is to resort to *output persistent* contracts; thus breaking such a symmetry.

[2] As for basic CCS [35] finite-stateness is an obvious consequence of the fact that the process algebra does not include static operators, like parallel or restriction.

Definition 7 (Output Persistence). *Consider the following notation:* $C \xrightarrow{\lambda}$ *means* $\exists C' : C \xrightarrow{\lambda} C'$ *and, given a (possibly empty) sequence of labels* $w = \lambda_1\lambda_2\cdots\lambda_{n-1}\lambda_n$*, we use* $C \xrightarrow{w} C'$ *to denote the sequence of transitions* $C \xrightarrow{\lambda_1} C_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_{n-1}} C_{n-1} \xrightarrow{\lambda_n} C'$*. A contract* C *is output persistent if, for any* C' *such that* $C \xrightarrow{w} C'$ *and* $C' \xrightarrow{\overline{a}_l}$*, the following holds:* $C' \xrightarrow{\surd}$ *and, if* $C' \xrightarrow{\alpha} C''$ *with* $\alpha \neq \overline{a}_l$*, then also* $C'' \xrightarrow{\overline{a}_l}$*.*

The output persistence property states that once a contract decides to execute an output, its actual execution is mandatory in order to successfully complete the execution of the contract. This property typically holds in languages for the description of service orchestrations (see e.g. WS-BPEL [40]) in which output actions cannot be used as guards in external choices (see e.g. the pick operator of WS-BPEL which is an external choice guarded on input actions).

The previous example of the authentication server (which is not output persistent) can be rephrased as follows to be used in a synchronous setting:

$$recX.(\ username.password.(\tau.\overline{accepted}_{client}.X + \tau.\overline{failed}_{client}.X) + \tau.\overline{updateAccounts}_{accountServer}.newAccounts.X\ )$$

Notice that, in this new version of the example, we have simply added an internal action τ in front of outputs occurring in choices. This guarantees that, at the moment the choice is to be resolved, the output action is not yet ready to be executed: it becomes available only after the τ and, then, its eventual execution is mandatory.

In the remainder, when we consider synchronous communication, we will restrict to output persistent contracts.

3.1 Synchronous Contract Composition

Synchronous systems are formed by the parallel composition of contracts.

Definition 8 (Synchronous Systems). *The syntax of synchronous systems is defined by the following grammar*

$$P \quad ::= \quad [C]_l \quad | \quad P \| P$$

We assume systems to be such that: (i) every contract subterm $[C]_l$ *occurs in* P *at a different location* l *and (ii) no output action with destination* l *is syntactically included inside a contract subterm occurring in* P *at the same location* l*, i.e. actions* $\overline{a}_l$ *cannot occur inside a subterm* $[C]_l$ *of* P*.*

A contract located at location l is denoted with $[C]_l$. Located contracts can be combined in parallel with the operator $P \| P$.

System operational semantics is defined by the rules in Table 4 plus symmetric rules. Transition system labels, still ranged over by λ, λ', $\cdots$, are now taken from the extended set $\{\overline{a}_{sr}, a_{sr}, \tau, \surd \mid a \in \mathcal{N}; s, r \in Loc\}$, where: $\overline{a}_{sr}$ (a_{sr}, resp.) denotes a potential output (input, resp.) with the sender being at location s and the receiver at location r; τ denotes a synchronization or a move performed internally by one contract in the system and $\surd$ denotes successful termination.

$$\frac{C \xrightarrow{\overline{a}_r} C'}{[C]_s \xrightarrow{\overline{a}_{sr}} [C']_s} \qquad \frac{C \xrightarrow{a} C'}{[C]_r \xrightarrow{a_{sr}} [C']_r} \qquad \frac{C \xrightarrow{\surd} C'}{[C]_l \xrightarrow{\surd} [C']_l}$$

$$\frac{P \xrightarrow{\lambda} P'}{P \| Q \xrightarrow{\lambda} P' \| Q} \lambda \neq \surd \qquad \frac{P \xrightarrow{\overline{a}_{sr}} P' \quad Q \xrightarrow{a_{sr}} Q'}{P \| Q \xrightarrow{\tau} P' \| Q'} \qquad \frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P \| Q \xrightarrow{\surd} P' \| Q'}$$

Table 4. Synchronous system semantics (symmetric rules omitted).

$$\frac{C \xrightarrow{\overline{a}_r} C'}{[C, \mathcal{Q}]_s \xrightarrow{\overline{a}_{sr}} [C', \mathcal{Q}]_s} \qquad [C, \mathcal{Q}]_r \xrightarrow{a_{sr}} [C, \mathcal{Q} :: a^s]_r \qquad \frac{C \xrightarrow{\surd} C'}{[C, \epsilon]_l \xrightarrow{\surd} [C', \epsilon]_l}$$

$$\frac{C \xrightarrow{a} C' \qquad b^l \in \mathcal{Q} \Rightarrow b \neq a}{[C, \mathcal{Q} :: a^s :: \mathcal{Q}']_r \xrightarrow{\tau} [C', \mathcal{Q} :: \mathcal{Q}']_r}$$

Table 5. Asynchronous system semantics (rules for parallel omitted).

3.2 Asynchronous Contract Composition

In asynchronous systems contracts are equipped with an input message queue.

Definition 9 (Asynchronous Systems). *The syntax of asynchronous systems is defined by the following grammar*

$$\begin{array}{lcl} P & ::= & [C, \mathcal{Q}]_l \quad | \quad P \| P \\ \mathcal{Q} & ::= & \epsilon \quad | \quad a^l :: \mathcal{Q} \end{array}$$

We assume asynchronous systems to be such that: (i) and (ii) of Definition 8 (with $[C, \mathcal{Q}]_l$ replacing $[C]_l$) hold true.

Terms $\mathcal{Q}$ denote message queues. They are sequences of messages, each one denoted with a^l where a is the action name and l is the location of the sender. We use "ϵ" to denote the empty message queue. Trailing ϵ are usually left implicit, and we use "::" also as an operator over the syntax: if $\mathcal{Q}$ and $\mathcal{Q}'$ are ϵ-terminated queues, according to the syntax above, then $\mathcal{Q} :: \mathcal{Q}'$ means appending the two queues into a single ϵ-terminated list. Therefore, if $\mathcal{Q}$ is a queue, then $\epsilon :: \mathcal{Q}$, $\mathcal{Q} :: \epsilon$, and $\mathcal{Q}$ are syntactically equal. In the following, when we talk about asynchronous contract systems, we will use the shorthand $[C]_l$ to stand for $[C, \epsilon]_l$.

Asynchronous system operational semantics is defined by the rules in Table 5 plus the rules for the parallel operator of Table 4. In Table 5 we assume that $b^l \in \mathcal{Q}$ holds true if and only if b^l syntactically occurs inside $\mathcal{Q}$. This notation is used in the premise of the novel τ synchronization rule that represents the consumption of an a message from the queue by removal of the oldest a one.

As an example consider the system: $[\overline{a}_s.\overline{b}_s]_r \parallel [b.a]_s$. After executing the two outputs, the system evolves to $[\mathbf{1}]_r \parallel [b.a, a^r :: b^r]_s$. The receiver is now ready to consume the two messages stored in the queue, thus reaching $[\mathbf{1}]_r \parallel [\mathbf{1}]_s$. Notice that the two messages are consumed in the opposite order of reception.

Notice that the information about the sender attached to queue messages is actually not used by the operational semantic rules in Table 5: even if omitted we would have obtained the same transitions. Nevertheless, we decided to use the same queue syntax as in [10] to be more adherent to reality, where messages can be distinguished e.g. depending on the sender. As a matter of fact, in [10], this information is used to produce, instead of τ actions, more informative labels that include denotation of the sender-receiver (this makes it possible to establish conformance w.r.t. a given choreographical specification).

3.3 Contract Refinement

We now recall the formal definition of independent contract refinement that preserves correct composition of contracts in both the synchronous and asynchronous cases. With $P \xrightarrow{\tau}{}^* P'$ we denote the existence of a (possibly empty) sequence of τ-labeled transitions starting from the system P and leading to P'.

Definition 10 (Correct Contract Composition – Compliance). *A system P is a correct contract composition, denoted $P\downarrow$, if for every P' such that $P \xrightarrow{\tau}{}^* P'$, there exists P'' such that $P' \xrightarrow{\tau}{}^* P''$ and $P'' \xrightarrow{\surd}$.*

Intuitively, a system composed of contracts is correct if any possible computation may guarantee completion, i.e. it can be extended to reach a successfully terminated computation (in the asynchronous case this means that all queues are empty). In this case, such contracts are called *compliant*. An example of contract composition that is correct (both in the synchronous and asynchronous case) is $[\overline{a}_{l_3}]_{l_1} \parallel [\overline{b}_{l_3}]_{l_2} \parallel [a.b]_{l_3}$. Another example is $[\overline{a}_s.\overline{b}_s]_r \parallel [b.a]_s$ considered above, which is correct only in the asychronous case.

We are now ready to define the notion of *contract refinement*. Given a contract C, we use $oloc(C)$ to denote the set of locations used as destinations in all the output actions occurring inside C.

Definition 11 (Independent Refinement). *A pre-order $\leq$ over contracts is an independent refinement if, for any $n \geq 1$, contracts $C_1, \ldots, C_n$ and $C'_1, \ldots, C'_n$ such that $\forall i.\, C'_i \leq C_i$, and distinguished location names $l_1, \ldots, l_n \in Loc$ such that $\forall i.\, oloc(C_i) \cup oloc(C'_i) \subseteq \{l_j \mid 1 \leq j \leq n \wedge j \neq i\}$, we have:*

$$([C_1]_{l_1} \parallel \ldots \parallel [C_n]_{l_n})\downarrow \;\Rightarrow\; ([C'_1]_{l_1} \parallel \ldots \parallel [C'_n]_{l_n})\downarrow$$

An independent refinement pre-order formalizes the possibility to replace in a correct contract composition every contract with one of its refinements, with the guarantee that the new system is still correct. In [6] it is shown that in the synchronous case, in the absence of the output persistence assumption, it

could happen that given two independent refinement pre-orders, their union is no longer an independent refinement pre-order. In other words, there exists no maximal independent refinement pre-order.

On the contrary, if we restrict to output persistent contracts or we consider asynchronous communication, we have that the maximal independent refinement pre-order exists: it can be achieved by considering a coarser form of refinement in which, given any system composed of a set of contracts, refinement is applied to one contract only (thus leaving the others unchanged). This form of refinement, that we call *compliance testing* [11], is a form of testing where both the test and the system under test must reach success. Given a system P, we use $loc(P)$ to denote the subset of Loc of the locations of contracts syntactically occurring inside P.

Definition 12 (Refinement Relation). *A contract C' is a refinement of a contract C denoted $C' \preceq C$, if and only if for all $l \in Loc$ and system P such that $l \notin loc(P)$ and $l \notin oloc(C) \cup oloc(C') \subseteq loc(P)$, we have:*

$$([C]_l \| P)\downarrow \;\Rightarrow\; ([C']_l \| P)\downarrow$$

Theorem 3 (Maximal Independent Refinement). *There exists a maximal independent refinement $\leq$ pre-order and it corresponds to the (compliance testing based) refinement relation "$\preceq$".*

3.4 Properties of Contract Refinement

We now discuss some properties of contract refinement and also show a sound characterization that is decidable for the synchronous case. We use $I(C)$ ($O(C)$, resp.) to stand for the subset of $\mathcal{N}$ of the names a of input (output, resp.) actions a ($\overline{a}$, resp.) syntactically occurring in C. Given $O \subseteq \mathcal{N}$ we assume $\overline{O}$ to stand for $\{\overline{a} \mid a \in O\}$.

We first observe that the refinement relation $\preceq$ allows input on new names (and unreachable outputs on new names) to be added in refined contracts.

Theorem 4 (Refinements with Extended Inputs and Outputs). *Let C, C' be contracts. Both of the following hold*

$$\begin{array}{lcl} C'\{\mathbf{0}/\alpha.C'' \mid \alpha \in I(C') - I(C)\} \preceq C & \Leftrightarrow & C' \preceq C \\ C'\{T/\alpha.C'' \mid \alpha \in \overline{O(C') - O(C)}\} \preceq C & \Leftrightarrow & C' \preceq C \end{array}$$

where T is: $\mathbf{0}$ in the synchronous case, $\tau.\mathbf{0}$ in the asynchronous case.

This theorem is a direct consequence of queue based communication (in the asynchronous case) and output persistence (in the synchronous case): a subcontract C' cannot perform reachable outputs that were not included in the potential outputs of the supercontract C; and, similarly, a compliant test P of a contract $[C]_l$ cannot perform reachable outputs directed to l that C cannot receive (e.g. in the asynchronous case $[a]_l \| [\overline{a}_l + \overline{b}_l]_{l'}$ is not a correct contract composition).

From this theorem we can derive two fundamental properties of the maximal independent refinement pre-order: external choices on inputs can be extended, e.g. $a + b \preceq a$; while internal choices on outputs can be reduced, e.g. $\overline{a}_l \preceq \overline{a}_l + \overline{b}_l$ in the asynchronous case and $\tau.\overline{a}_l \preceq \tau.\overline{a}_l + \tau.\overline{b}_l$ in the synchronous one, because the lefthand term is more deterministic (typical property in testing).

We now focus on determining an algorithmic sound characterization of the synchronous contract refinement relation. This is achieved by resorting to the theory of fair testing, called *should-testing* [41]. As a side result we also have that the refinement relation $\preceq$ is coarser than fair testing preorder. We denote with $\preceq_{test}$ the *should-testing* pre-order defined in [41] where we consider $\surd$ to be included in the set of actions of terms under testing as any other action ($\surd$ is treated as a normal action and not as the special action representing success of tests in [41]). In order to resort to the theory should-testing, we define a normal form for contracts C, denoted with $\mathcal{NF}(C)$, that corresponds to terms of the language in [41] (mainly a matter of replacing $\mathbf{1}$ with a $\surd$ action, see [8] for details).

Theorem 5 (Resorting to Fair Testing). *Let C, C' be contracts. We have*

$$\mathcal{NF}(C'\{\mathbf{0}/\alpha.C'' \mid \alpha \in I(C') - I(C)\}) \preceq_{test} \mathcal{NF}(C) \quad \Rightarrow \quad C' \preceq C$$

The opposite implication does not hold in general. This can be easily seen by considering *uncontrollable* contracts, i.e. contracts for which there is no compliant test. For instance the contract $\mathbf{0}$, any other contract $a.b.\mathbf{0}$ or $c.d.\mathbf{0}$ or more complex examples like $a + a.b$. These contracts are all equivalent according to our refinement relation, but of course not according to fair testing. Notice that such uncontrollable contracts have completely different traces: this means that trace pre-order is *not* coarser than our refinement relation.

4 Session Types

In this section we move to session types, in particular we report about our study of asynchronous session subtyping. Session types [28,29] are types for controlling the communication behaviour of processes over channels. In a very simple but effective way, they express the pattern of sends and receives that a process must perform. They are, therefore, similar to behavioural contracts, but more constrained in the kind of behaviours they can express. Since they can guarantee freedom from some basic programming errors, session types are becoming popular with many main stream language implementations, e.g., Haskell [34], Go [39] or Rust [30]. In [38] session subtyping is introduced for asynchronous communication and it is also stated that it is decidable. Recently it has been proven that, on the contrary, it is undecidable. Here we present such an undecidability result [4] and the decidability result in [5], where the largest known decidable fragment is introduced. In particular, we recall the basic definitions of session types and synchronous and asynchronous session subtyping. We then report the undecidability proof in [4]. Finally, we present the fragment of *single-out* (and *single-in*) session types, for which we show asynchronous subtyping to

be decidable [5]. The techniques for these (un)decidability results can be seen as improvements of those developed for Linda process calculi: reduction from Turing complete computational models and exploitation of well quasi orderings.

4.1 Session Subtyping

Session subtyping, which is the counterpart for session types of refinement for behavioural contracts, was first introduced by Gay and Hole [26] for a session-based π-calculus where communication is synchronous. Session subtyping of [26] is endowed with covariant/contravariant properties that correspond to those we observed on behavioural contract refinement: internal choices on outputs can be reduced, while external choices on inputs can be extended. To the best of our knowledge, Mostrous et al. [38] were the first to adapt the notion of session subtyping to an asynchronous setting. Their computation model is a session π-calculus with asynchronous communication that makes use of session queues for maintaining the order in which messages are sent. Based on such a model they introduce the idea of *output anticipation*, which is also a main feature of our theory in [4,5] that we present here. Mostrous and Yoshida [37] extended the notion of asynchronous subtyping to session types for the higher-order π-calculus. They also observed that their definition of asynchronous subtyping allows for *orphan messages*, i.e. sent messages which are never consumed from the session queue. Orphan messages are, instead, prohibited with the definition of subtyping given by Chen et al. [20]: they show that such a definition is both sound and complete w.r.t. type safety and orphan message freedom.

We start with the formal syntax of binary session types, adopting a simplified notation (used, e.g., in [4,5]) without dedicated constructs for sending an output/receiving an input. We instead represent outputs and inputs directly inside choices. More precisely, we consider output selection $\oplus\{l_i : T_i\}_{i\in I}$, expressing an internal choice among outputs, and input branching $\&\{l_i : T_i\}_{i\in I}$, expressing an external choice among inputs. Each possible choice is labeled by a label l_i, taken from a global set of labels L, followed by a session continuation T_i. Labels in a branching/selection are assumed to be pairwise distinct.

Definition 13 (Session Types). *Given a set of labels L, ranged over by l, the syntax of binary session types is given by the following grammar:*

$$T \quad ::= \quad \oplus\{l_i : T_i\}_{i\in I} \quad | \quad \&\{l_i : T_i\}_{i\in I} \quad | \quad \mu\mathbf{t}.T \quad | \quad \mathbf{t} \quad | \quad \mathbf{end}$$

A session type is single-out *if, for all of its subterms $\oplus\{l_i : T_i\}_{i\in I}$, $|I| = 1$; it is* single-in *if, for all of its subterms $\&\{l_i : T_i\}_{i\in I}$, $|I| = 1$.*

In the sequel, we leave implicit the index set $i \in I$ in input branchings and output selections when it is already clear from the denotation of the types. Note also that we abstract from the type of the message that could be sent over the channel, since this is orthogonal to our theory. Types $\mu\mathbf{t}.T$ and $\mathbf{t}$ denote standard tail recursion for recursive types. We assume recursion to be guarded: in $\mu\mathbf{t}.T$, the recursion variable $\mathbf{t}$ occurs within the scope of an output or an input type.

In the following, we will consider closed terms only, i.e., types with all recursion variables $\mathbf{t}$ occurring under the scope of a corresponding definition $\mu\mathbf{t}.T$. Type **end** denotes the type of a channel that can no longer be used.

For session types, we define the usual notion of duality: given a session type T, its dual $\overline{T}$ is defined as: $\overline{\oplus\{l_i : T_i\}_{i\in I}} = \&\{l_i : \overline{T}_i\}_{i\in I}$, $\overline{\&\{l_i : T_i\}_{i\in I}} = \oplus\{l_i : \overline{T}_i\}_{i\in I}$, $\overline{\mathbf{end}} = \mathbf{end}$, $\overline{\mathbf{t}} = \mathbf{t}$, and $\overline{\mu\mathbf{t}.T} = \mu\mathbf{t}.\overline{T}$. In the sequel, we say that a relation $\mathcal{R}$ on session types is *dual closed* if $(S,T) \in \mathcal{R}$ implies $(\overline{T},\overline{S}) \in \mathcal{R}$.

We start by considering a *synchronous* subtyping relation, similar to that of Gay and Hole [26] but, to be more consistent with contracts, following a process-oriented instead of a channel-based approach.[3] Moreover, following [38], we consider a generalized version of unfolding that allows us to unfold recursions $\mu\mathbf{t}.T$ as many times as needed.

Definition 14 (n-unfolding).

$$\begin{array}{ll}
\mathsf{unfold}^0(T) = T & \mathsf{unfold}^1(\oplus\{l_i : T_i\}_{i\in I}) = \oplus\{l_i : \mathsf{unfold}^1(T_i)\}_{i\in I} \\
\mathsf{unfold}^1(\mu\mathbf{t}.T) = T\{\mu\mathbf{t}.T/\mathbf{t}\} & \mathsf{unfold}^1(\&\{l_i : T_i\}_{i\in I}) = \&\{l_i : \mathsf{unfold}^1(T_i)\}_{i\in I} \\
\mathsf{unfold}^1(\mathbf{end}) = \mathbf{end} & \mathsf{unfold}^n(T) = \mathsf{unfold}^1(\mathsf{unfold}^{n-1}(T))
\end{array}$$

Definition 15 (Synchronous Subtyping, $\leq_s$). *$\mathcal{R}$ is a synchronous subtyping relation whenever $(T,S) \in \mathcal{R}$ implies that:*

1. *if $T = \mathbf{end}$ then $\exists n \geq 0$ such that $\mathsf{unfold}^n(S) = \mathbf{end}$;*
2. *if $T = \oplus\{l_i : T_i\}_{i\in I}$ then $\exists n \geq 0$ such that $\mathsf{unfold}^n(S) = \oplus\{l_j : S_j\}_{j\in J}$, $I \subseteq J$ and $\forall i \in I.\,(T_i, S_i) \in \mathcal{R}$;*
3. *if $T = \&\{l_i : T_i\}_{i\in I}$ then $\exists n \geq 0$ such that $\mathsf{unfold}^n(S) = \&\{l_j : S_j\}_{j\in J}$, $J \subseteq I$ and $\forall j \in J.\,(T_j, S_j) \in \mathcal{R}$;*
4. *if $T = \mu\mathbf{t}.T'$ then $(T'\{T/\mathbf{t}\}, S) \in \mathcal{R}$.*

T is a synchronous subtype of S, written $T \leq_s S$, if there is a synchronous subtyping relation $\mathcal{R}$ such that $(T,S) \in \mathcal{R}$.

Two types T and S are related by $\leq_s$, whenever S is able to simulate T with output and input types enjoying covariance and contravariance properties, respectively. Notice the asymmetric use of unfolding between the left- and right-hand terms T and S: in T recursion is always unfolded once, while in S many unfoldings can be needed in order to expose the starting operator of T.

As already discussed, subtyping is the counterpart of contract refinement in the context of session types. Consider, for instance,

$$\&\{a : \mathbf{end}, b : \mathbf{end}\} \leq_s \&\{a : \mathbf{end}\} \qquad \oplus\{a : \mathbf{end}\} \leq_s \oplus\{a : \mathbf{end}, b : \mathbf{end}\}$$

that hold for input contravariance and output covariance. These examples of subtypings precisely correspond to those of contract refinements commented in Section 3.4. Note that, while in the case of contracts they were obtained as a

[3] Differently from our definitions, in the channel-based approach of Gay and Hole [26] subtyping is covariant on branchings and contra-variant on selections.

consequence of considering the maximal independent refinement, in the theory of session types they are taken by definition.

We now consider the standard notion of *asynchronous* subtyping $\leq$ introduced by Chen et al. [20], which enjoys orphan message freedom; we consider the simple rephrasing based on dual closeness we introduced in [5]. In the definition of $\leq$ we use the following notion of input context.

Definition 16 (Input Context). *An input context $\mathcal{A}$ is a session type with multiple holes defined by the syntax:* $\mathcal{A} ::= \quad []^n \quad | \quad \&\{l_i : \mathcal{A}_i\}_{i \in I}$. *The holes $[]^n$, with $n \in \mathbb{N}^+$, of an input context $\mathcal{A}$ are assumed to be consistently enumerated, i.e. there exists $m \geq 1$ such that $\mathcal{A}$ includes one and only one $[]^n$ for each $n \leq m$. Given types $T_1, \ldots, T_m$, we use $\mathcal{A}[T_k]^{k \in \{1,\ldots,m\}}$ to denote the type obtained by filling each hole k in $\mathcal{A}$ with the corresponding term T_k.*

Definition 17 (Asynchronous Subtyping, $\leq$). *$\mathcal{R}$ is an asynchronous subtyping relation whenever it is* dual closed *and $(T, S) \in \mathcal{R}$ implies* 1., 3., *and* 4. *of Definition 15, plus the following modified version of* 2.*:*

2. *if $T = \oplus\{l_i : T_i\}_{i \in I}$ then $\exists n \geq 0, \mathcal{A}$ such that*
   - $\mathsf{unfold}^n(S) = \mathcal{A}[\oplus\{l_j : S_{kj}\}_{j \in J_k}]^{k \in \{1,\ldots,m\}}$,
   - $\forall k \in \{1, \ldots, m\}. I \subseteq J_k$ *and*
   - $\forall i \in I, (T_i, \mathcal{A}[S_{ki}]^{k \in \{1,\ldots,m\}}) \in \mathcal{R}$

T is an asynchronous subtype of S, written $T \leq S$, if there is an asynchronous subtyping relation $\mathcal{R}$ such that $(T, S) \in \mathcal{R}$.

We now explain the modified version of Rule 2. and its impact on the obtained subtyping relation. Concerning the adopted notation, for each hole k of the input context $\mathcal{A}$ (which is at the beginnig of the righthand term S after any needed unfolding), we take l_j, with $j \in J_k$, to be the labels of the output selection in the hole. Moreover, we use S_{kj} to denote the type reached after output l_j in the hole k. An important characteristic of asynchronous subtyping (formalized by Rule 2. above) is the following one. In a subtype output selections can be anticipated so to bring them before the input branchings that in the supertype occur in front of them. For example

$$\oplus\big\{l : \&\{l_1 : T_1, l_2 : T_2\}\big\} \;\leq\; \&\big\{l_1 : \oplus\{l : T_1\}, l_2 : \oplus\{l : T_2\}\big\}$$

where the output selection with label l (occurring in the supertype) is anticipated w.r.t. the input branching with labels l_1 and l_2 (such an output selection is present in *all* its input branches). As already discussed in the Introduction, output anticipation reflects the fact that we are considering asynchronous communication protocols in which messages are stored in queues. In this setting, it is safe to replace a peer that follows a given protocol with another one following a modified protocol where outputs are anticipated: in fact, the difference is simply that such outputs will be stored earlier in the communication queue.

As a further example, consider the types $T = \mu\mathbf{t}.\&\{l : \oplus\{l : \mathbf{t}\}\}$ and $S = \mu\mathbf{t}.\&\{l : \&\{l : \oplus\{l : \mathbf{t}\}\}\}$. We have $T \leq S$ by considering an infinite subtyping

relation including pairs (T', S'), with S' being $\&\{l : S\}$, $\&\{l : \&\{l : S\}\}$, $\&\{l : \&\{l : \&\{l : S\}\}\}$, ...; that is, the effect of each output anticipation is that a new input $\&\{l : \_\}$ is accumulated in the initial part of the r.h.s. It is worth to observe that every accumulated input $\&\{l : \_\}$ is eventually consumed in the simulation game (orphan message freedom), but the accumulated inputs grows unboundedly.

4.2 Undecidability of Asynchronous Subtyping

The proof of undecidability of the asynchronous subtyping relation, taken from [4], is by reduction from the acceptance problem for queue machines.

Definition 18 (Queue machine). *A queue machine M is defined by a six-tuple $(Q, \Sigma, \Gamma, \$, s, \delta)$ where: Q is a finite set of states; $\Sigma \subset \Gamma$ is a finite set denoting the input alphabet; Γ is a finite set denoting the queue alphabet (ranged over by A, B, C, X); $\$ \in \Gamma - \Sigma$ is the initial queue symbol; $s \in Q$ is the start state; $\delta : Q \times \Gamma \to Q \times \Gamma^*$ is the transition function.*

We now formally define queue machine computations.

Definition 19 (Queue machine computation). *A* configuration *of a queue machine is an ordered pair (q, γ) where $q \in Q$ is its* current state *and $\gamma \in \Gamma^*$ is the* queue *(Γ^* is the Kleene closure of Γ). The starting configuration on an input string x is $(s, x\$)$. The transition relation $\to_M$ over configurations $Q \times \Gamma^*$, leading from a configuration to the next one, is defined as follows. For any $p, q \in Q$, $A \in \Gamma$ and $\alpha, \gamma \in \Gamma^*$ we have $(p, A\alpha) \to_M (q, \alpha\gamma)$ whenever $\delta(p, A) = (q, \gamma)$. A machine M accepts an input x if it eventually terminates on input x, i.e. it reaches a blocking configuration with the empty queue (notice that, as the transition relation is total, the unique way to terminate is by emptying the queue). Formally, x is accepted by M if $(s, x\$) \to_M^* (q, \epsilon)$ where ϵ is the empty string and $\to_M^*$ is the reflexive and transitive closure of $\to_M$.*

Queue machines are Turing complete, see [31] (page 354) and [4].

Our goal is to construct a pair of types, say T and S, from a given queue machine M and a given input x, such that: $T \leq S$ if and only if x is not accepted by M. Intuitively, type T encodes the finite control of M, i.e., its transition function δ, starting from its initial state s. And type S encodes the machine queue that initially contains $x\$$, where x is the input string $x = X_1 \cdots X_n$ of length $n \geq 0$. The set of labels L for such types T and S is M's queue alphabet Γ.

Formally, the queue of a machine is encoded into a session type as follows:

Definition 20 (Queue Encoding). *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine and let $C_1 \cdots C_m \in \Gamma^*$, with $m \geq 0$. We define:*

$$[\![C_1 \cdots C_m]\!] = \&\{C_1 : \ldots \&\{C_m : \mu \mathbf{t}. \oplus \{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma}\}\}$$

Given a configuration (q, γ) of M, the encoding of the queue $\gamma = C_1 \cdots C_m$ is thus defined as $[\![C_1 \cdots C_m]\!]$.

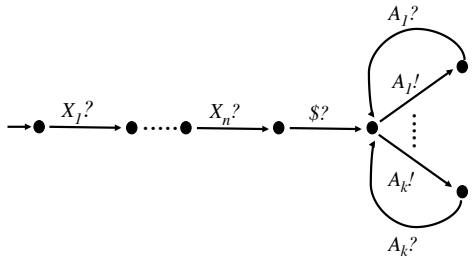


Fig. 1. Session type encoding the initial queue $X_1 \cdots X_n\$$

Note that whenever $m = 0$, we have $[\![\epsilon]\!] = \mu \mathbf{t}. \oplus \{A : \&\{A : \mathbf{t}\}\}_{A \in \Gamma}$. Observe that we are using a slight abuse of notation: in both output selections and input branchings, labels l_A, with $A \in \Gamma$, are simply denoted by A.

Figure 1 contains a graphical representation of the queue encoding with its initial content $X_1 \cdots X_n\$$. In order to better clarify our development, we graphically represent session types as labeled transition systems (in the form of communicating automata [3]), where an output selection $\oplus\{l_i : T_i\}_{i \in I}$ is represented as a choice among alternative output transitions labeled with "$l_i!$", and an input branching $\&\{l_i : T_i\}_{i \in I}$ is represented as a choice among alternative input transitions labeled with "$l_i?$". Intuitively, we encode a queue containing symbols $C_1 \cdots C_m$ with a session type that starts with m inputs with labels C_1, ..., C_m, respectively. Thus, in Figure 1, we have $C_1 \cdots C_m = X_1 \cdots X_n\$$. After such sequence of inputs, representing the current queue content, there is a recursive type representing the capability to enqueue new symbols. Such a type repeatedly performs an output selection with one choice for each symbol A_i in the queue alphabet Γ (with k being the cardinality of Γ), followed by an input labeled with the same symbol A_i.

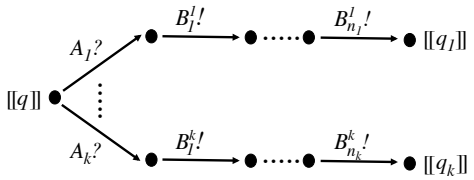
We now give the definition of the type modelling the finite control of a queue machine, i.e., the encoding of the transition function δ.

Definition 21 (Finite Control Encoding). *Let $M = (Q, \Sigma, \Gamma, \$, s, \delta)$ be a queue machine and let $q \in Q$ and $\mathcal{S} \subseteq Q$. We define:*

$$[\![q]\!]^{\mathcal{S}} = \begin{cases} \mu \mathbf{q}.\&\{A : \oplus\{B_1^A : \cdots \oplus \{B_{n_A}^A : [\![q']\!]^{\mathcal{S} \cup q}\}\}\}_{A \in \Gamma} \\ \qquad \text{if } q \notin \mathcal{S} \text{ and } \delta(q, A) = (q', B_1^A \cdots B_{n_A}^A) \\ \mathbf{q} \qquad \text{if } q \in \mathcal{S} \end{cases}$$

The encoding of the transition function of M is then defined as $[\![s]\!]^{\emptyset}$.

The encoding of the finite control is a recursively defined term with one recursion variable $\mathbf{q}$ for each state $q \in Q$ of the machine. Above, $[\![q]\!]^{\mathcal{S}}$ is a function that, given a state q and a set of states $\mathcal{S}$, returns a type representing the possible behaviour of the queue machine starting from state q. Such behaviour consists of first reading from the queue (input branching on $A \in \Gamma$) and then writing on the queue a sequence of symbols $B_1^A, \ldots, B_{n_A}^A$. The parameter $\mathcal{S}$ is necessary for



(for $\Gamma = \{A_i | i \leq k\}$ and $\delta(q, A_i) = (q_i, B^i_1 \cdots B^i_{n_i})$ for every i)

Fig. 2. Session type encoding a finite control.

managing the recursive definition of this type. In fact, as the definition of the encoding function is itself recursive, this parameter keeps track of the states that have been already encoded (see example below). In Figure 2, we report a graphical representation of the Labelled Transition System corresponding to the session type that encodes the queue machine finite control, i.e. the transition function δ. Each state $q \in Q$ is mapped onto a state $[\![q]\!]$ of a session type, which performs an input branching with a choice for each symbol in the queue alphabet Γ (with k being the cardinality of Γ). Each of these choices represents a possible symbol that can be read from the queue. After this initial input branching, each choice continues with a sequence of outputs labeled with the symbols that are to be inserted in the queue (after the symbol labeling that choice has been consumed). This is done according to function δ, assuming that $\delta(q, A_i) = (q_i, B^i_1 \cdots B^i_{n_i})$, with $n_i \geq 0$, for all i in $\{1, \ldots, k\}$. After the insertion phase, state $[\![q_i]\!]$ of the session type corresponding to state q_i of the queue machine is reached.

Notice that, queue insertion actually happens in the encoding because, when the encoding of the finite control performs an output of a B symbol, the encoding of the queue must mimic such an output, possibly by anticipating it. This has the effect of adding an input on B at the end of the sequence of initial inputs of the queue machine encoding.

Theorem 6. *Given a queue machine $M = (Q, \Sigma, \Gamma, \$, s, \delta)$, an input string x, and the two types $T = [\![s]\!]^{\emptyset}$ and $S = [\![x\$]\!]$, we have that M accepts x iff $T \not\leq S$.*

4.3 Decidability of Single-Out/Single-In Asynchronous Subtyping

We now show decidability of asynchronous session subtyping over single-out/ single-in session types. The full technical machinery can be found in [5].

We start by giving a procedure (an algorithm that does not necessarily terminate) for the general subtyping relation, which we showed to be undecidable. Such a procedure is inspired by the one proposed by Mostrous et al. [38] for asynchronous subtyping in multiparty session types. In order to do so, we introduce two functions on the syntax of types. The function $\mathsf{outDepth}$ calculates how many unfolding are necessary for bringing an output outside a recursion. If that is not possible, the function is undefined (denoted by $\perp$). As an example consider, for any T_1 and T_2, $\mathsf{outDepth}(\oplus\{l_1 : T_1,\ l_2 : T_2\}) = 0$. On the other hand, consider

the type $T_{ex} = \&\Big\{l_1 : \mu\mathbf{t}.\oplus\{l_2 : T_1\},\ l_3 : \mu\mathbf{t}.\&\{l_4 : \mu\mathbf{t}'.\oplus\{l_5 : T_2\}\}\Big\}$: we have, $\mathsf{outDepth}(T_{ex}) = 2$. We then define $\mathsf{outUnf}()$, a variant of the unfolding function given in Definition 14, which unfolds only where it is necessary, in order to reach an output. The function above differs from unfold^n: for example, $\mathsf{unfold}^2(T_{ex})$ would unfold twice both subterms $\mu\mathbf{t}.\oplus\{l_2 : T_1\}$ and $\mu\mathbf{t}.\&\{l_4 : \mu\mathbf{t}'.\oplus\{l_5 : T_2\}\}$. On the other hand, applying $\mathsf{outDepth}$ to the same term would unfold once the term reached with l_1 and twice the one reached with l_3. In the subtyping procedure defined below we make use of $\mathsf{outUnf}()$ in order to have that recursive definitions under the scope of an output are never unfolded. This guarantees that during the execution of the procedure, even if the set of reached terms could be unbounded, all the subterms starting with an output are taken from a bounded set of terms. This is important to guarantee termination of the algorithm that we are going to define as an extension of the procedure described below.

Subtyping Procedure. An environment Σ is a set containing pairs (T, S), where T and S are types. Judgements are triples of the form $\Sigma \vdash T \leq_{\mathsf{a}} S$ which intuitively read as "in order to succeed, the procedure must check whether T is a subtype of S, provided that pairs in Σ have already been visited". Our *subtyping procedure*, applied to the types T and S, consists of deriving the state space of our judgments using the rules in Figure 3 bottom-up starting from the initial judgement $\emptyset \vdash T \leq_{\mathsf{a}} S$. More precisely, we use the transition relation $\Sigma \vdash T \leq_{\mathsf{a}} S \to \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ to indicate that if $\Sigma \vdash T \leq_{\mathsf{a}} S$ matches the conclusions of one of the rules in Figure 3, then $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is produced by the corresponding premises. The procedure explores the reachable judgements according to this transition relation. We give highest priority to rule Asmp, thus ensuring that at most one rule is applicable.[4] The idea behind Σ is to avoid cycles when dealing with recursive types. Rules RecR_1 and RecR_2 deal with the case in which the type on the right-hand side is a recursion and must be unfolded. If the type on the left-hand side is not an output then the procedure simply adds the current pair to Σ and continues. On the other hand, if an output must be found, we apply RecR_1 which checks whether such output is available. Rule Out allows nested outputs to be anticipated (when not under recursion) and condition $\big(\mathcal{A} \neq [\,]^1\big) \Rightarrow \forall i \in I.\& \in T_i$ (inspired by [20]) makes sure there are no orphan messages. In fact, this condition implies that if there is some output which is anticipated in the subtype w.r.t. some inputs, in every continuation of the subtype there are input actions that will eventually reproduce also the input behaviour of the supertype. The remaining rules are self-explanatory. $\Sigma \vdash T \leq_{\mathsf{a}} S \to^* \Sigma' \vdash T' \leq_{\mathsf{a}} S'$ is the reflexive and transitive closure of the transition relation among judgements. We write $\Sigma \vdash T \leq_{\mathsf{a}} S \to_{\mathsf{ok}}$ if the judgement $\Sigma \vdash T \leq_{\mathsf{a}} S$ matches the conclusion of one of the axioms Asmp or End, and $\Sigma \vdash T \leq_{\mathsf{a}} S \to_{\mathsf{err}}$ to mean that no rule can be applied to $\Sigma \vdash T \leq_{\mathsf{a}} S$. Due to input branching and output selection, the rules In and Out could generate branching also in the state space

[4] The priority of Asmp is sufficient because all the other rules are alternative, i.e., given a judgement $\Sigma \vdash T \leq_{\mathsf{a}} S$ there are no two rules different from Asmp that can be both applied.

$$\frac{(\mathcal{A} \neq [\,]^1) \Rightarrow \forall i \in I.\& \in T_i \quad \forall n.I \subseteq J_n \qquad \forall i \in I\,.\,\Sigma \vdash T_i \leq_{\mathsf{a}} \mathcal{A}[S_{ni}]^n}{\Sigma \vdash \oplus\{l_i : T_i\}_{i\in I} \leq_{\mathsf{a}} \mathcal{A}[\oplus\{l_j : S_{nj}\}_{j\in J_n}]^n}\ \mathsf{Out}$$

$$\frac{J \subseteq I \quad \forall j \in J\,.\,\Sigma \vdash T_j \leq_{\mathsf{a}} S_j}{\Sigma \vdash \&\{l_i : T_i\}_{i\in I} \leq_{\mathsf{a}} \&\{l_j : S_j\}_{j\in J}}\ \mathsf{In} \qquad \frac{}{\Sigma \vdash \mathbf{end} \leq_{\mathsf{a}} \mathbf{end}}\ \mathsf{End}$$

$$\frac{}{\Sigma, (T, S) \vdash T \leq_{\mathsf{a}} S}\ \mathsf{Asmp} \qquad \frac{\Sigma, (\mu\mathbf{t}.T, S) \vdash T\{\mu\mathbf{t}.T/\mathbf{t}\} \leq_{\mathsf{a}} S}{\Sigma \vdash \mu\mathbf{t}.T \leq_{\mathsf{a}} S}\ \mathsf{RecL}$$

$$\frac{T = \mathbf{end} \vee T = \&\{l_i : T_i\}_{i\in I} \quad \Sigma, (T, \mu\mathbf{t}.S) \vdash T \leq_{\mathsf{a}} S\{\mu\mathbf{t}.S/\mathbf{t}\}}{\Sigma \vdash T \leq_{\mathsf{a}} \mu\mathbf{t}.S}\ \mathsf{RecR}_1$$

$$\frac{\mathsf{outDepth}(S) \geq 1 \quad \Sigma, (\oplus\{l_i : T_i\}_{i\in I}, S) \vdash \oplus\{l_i : T_i\}_{i\in I} \leq_{\mathsf{a}} \mathsf{outUnf}(S)}{\Sigma \vdash \oplus\{l_i : T_i\}_{i\in I} \leq_{\mathsf{a}} S}\ \mathsf{RecR}_2$$

Fig. 3. A Procedure for Checking Subtyping

to be explored by the procedure. Namely, given a judgement $\Sigma \vdash T \leq_{\mathsf{a}} S$, there are several subsequent judgements $\Sigma' \vdash T' \leq_{\mathsf{a}} S'$ such that $\Sigma \vdash T \leq_{\mathsf{a}} S \rightarrow \Sigma' \vdash T' \leq_{\mathsf{a}} S'$. The procedure could (i) successfully terminate because all the explored branches reach a successful judgement $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{ok}}$, (ii) terminate with an error in case at least one judgement $\Sigma' \vdash T' \leq_{\mathsf{a}} S' \rightarrow_{\mathsf{err}}$ is reached, or (iii) diverge because no branch terminates with an error and at least one branch never reaches a succesful judgement. As we prove in [5] the procedure is sound with respect to asynchronous subtyping $\leq$ and it can diverge only if the checked types are in the $\leq$ relation.

If we consider types T and S of the example considered after Definition 17 the subtyping procedure in Figure 3 applied to $\emptyset \vdash T \leq_{\mathsf{a}} S$ does not terminate. The problem is that the termination rule Asmp cannot be applied because the term on the r.h.s. (i.e. the supertype) generates always new terms in the form $\&\{l : \&\{l : \ldots \&\{l : S\} \ldots\}\}$. Notice that, in this particular example, these infinitely many distinct terms are obtained by adding single inputs (i.e. single-choice input branchings) in front of the term in the r.h.s.: we call this *linear input accumulation*. In general, however, input accumulation takes the *form of a tree* (thus accounting for all possible alternative accumulated input behaviors at the same time).

We now show how to decide asynchronous subtyping over single-out types, i.e. when input accumulation can indeed be in the general form of a tree, but, due to the absence of output selections with multiple choices, it gets accumulated in a deterministic (i.e. unique) way. This will also allow us to deal with single-in types by exploiting duality. As anticipated, it deals with general input accumulation by representing it as a tree. We need to be able to extract the leaves from these trees: this is done by the *leaf set* function. The leaf set of a session type T is the set of subterms reachable from its root through a path of inputs. For example,

the leaf set of the term $\&\{l_1 : \mu\mathbf{t}. \oplus \{l_2 : \mathbf{t}\}, l_3 : \&\{l_4 : \oplus\{l_2 : \mu\mathbf{t}. \oplus \{l_2 : \mathbf{t}\}\}\}\}$ is $\{\mu\mathbf{t}. \oplus \{l_2 : \mathbf{t}\}, \oplus\{l_2 : \mu\mathbf{t}. \oplus \{l_2 : \mathbf{t}\}\}$.

During the check of subtyping, according to Figure 3 (rule Out), when a term in the r.h.s. having input accumulation has to mimic an output in front of the l.h.s., such output must be present in front of all the leaves of the tree. In this case, the checking continues by anticipating the output from all the leaves. We make use of an auxiliary *output anticipation* function, called antOut, that indicates the way a term changes after having anticipated a sequence of outputs. $\mathsf{antOut}(T, \tilde{l})$ yields the term obtained from T by anticipating all outputs occurring in the sequence $\tilde{l}$. For example, the function applied to the type $T = \mu\mathbf{t}. \oplus \{l_1 : \&\{l : \oplus\{l_2 : \mathbf{t}\}, l' : \oplus\{l_2 : \mathbf{t}\}\}$ and the sequence (l_1, l_2) returns $\&\{l : T, l' : T\}$, while it is undefined with the sequence (l_1, l_1). Moreover, we say that T can infinitely anticipate outputs, written $\mathsf{antOutInf}(T)$, if there exists an infinite sequence of labels $l_{i_1} \cdots l_{i_j} \cdots$ such that $\mathsf{antOut}(T, l_{i_1} \cdots l_{i_n})$ is defined for every n. The definition of $\mathsf{antOutInf}(T)$ is not algorithmic in that it quantifies on every possible natural number n. Nevertheless, it can be decided by checking whether, for every session type obtained from T by means of output anticipations, all the terms populating its leaf set can anticipate the same output label. Although the types that can be obtained from T by means of output anticipations may be infinite, the terms populating the leaf sets are finite and are over-approximated by the function $\mathsf{reach}(T)$ which is defined as the minimal set of (single-out) session types such that:

1. $T \in \mathsf{reach}(T)$;
2. $\&\{l_i : T_i\}_{i \in I} \in \mathsf{reach}(T)$ implies $T_i \in \mathsf{reach}(T)$ for every $i \in I$;
3. $\mu\mathbf{t}.T' \in \mathsf{reach}(T)$ implies $T'\{\mu\mathbf{t}.T'/\mathbf{t}\} \in \mathsf{reach}(T)$;
4. $\oplus\{l : T'\} \in \mathsf{reach}(T)$ implies $T' \in \mathsf{reach}(T)$.

Notice that $\mathsf{reach}(T)$ contains the session types obtained by consuming initial inputs and outputs, and by unfolding recursion when it is at the top level.

Proposition 3. *Given a single-out session type T, $\mathsf{reach}(T)$ is finite and it is decidable whether $\mathsf{antOutInf}(T)$.*

Subtyping algorithm for single-out types. We are now ready to present an additional termination condition that, once included into the subtyping procedure in Figure 3, makes it a valid algorithm for checking subtyping for single-out types. The termination condition is defined as an additional rule, named $\mathsf{Asmp2}$, that complements the already defined Asmp rule by detecting those cases in which the subtyping procedure in Figure 3 does not terminate ($\mathsf{Asmp2}$, presented below, is assumed to have the same priority as rule Asmp: both rules have highest priority). The new rule is defined parametrically on the session type Z, which is the type on the right-hand side of the initial pair of types to be checked (i.e. the algorithm is intended to check $V \leq Z$, for some type Z). We start from the initial judgement $\emptyset \vdash V \leq_{\mathbf{t}} Z$ and then apply from bottom to top the rules in

Figure 3, where $\leq_{\mathsf{a}}$ is replaced by $\leq_{\mathsf{t}}$, plus the following additional rule:

$$\frac{S \in \mathsf{reach}(Z) \qquad \mathsf{antOutInf}(S) \qquad |\gamma| < |\beta| \qquad \mathsf{leafSet}(\mathsf{antOut}(S,\gamma)) = \mathsf{leafSet}(\mathsf{antOut}(S,\beta))}{\Sigma, (T, \mathsf{antOut}(S,\gamma)) \vdash T \leq_{\mathsf{t}} \mathsf{antOut}(S,\beta)} \ \mathsf{Asmp2}$$

Intuitively, we have that this additional termination rule guarantees to catch all those cases where the term on the right grows indefinitely, by anticipating outputs and accumulating inputs. These infinitely many distinct types are anyway obtainable starting from the finite set $\mathsf{reach}(Z)$, by means of output anticipations. Hence there exists $S \in \mathsf{reach}(Z)$ that can generate infinitely many of these types: this guarantees $\mathsf{antOutInf}(S)$ to be true. As observed above, the leaves of such infinitely many terms are themselves taken from the finite set $\mathsf{reach}(Z)$. In Section 2 we have commented that multiset inclusion is a wqo over multisets defined on a finite domain; here we use a similar result according to which set equality is a wqo over the subsets of a finite given set. In fact, the possible subsets in this case are finite thus in an infinite sequence of such subsets at least one is repeated. The termination of our algorithm follows this wqo: Asmp2, besides checking conditions that are guaranteed to hold if the procedure $\leq_{\mathsf{a}}$ continues indefinitely, checks also for the equality between the set of leaves of the r.h.s. term in the current judgement and in a previously checked one.

Theorem 7 (Decidability for Single-out Types). *Asynchronous subtyping $\leq$ over single-out session types is decidable.*

Exploiting dual closeness of $\leq$ we can use the algorithm presented for single-out types also for single-in types (it is sufficient to check subtyping on the duals, observing that the dual of a single-in type is single-out).

Corollary 1 (Decidability for Single-in Types). *Asynchronous subtyping $\leq$ over single-in session types is decidable.*

5 Conclusion

In this survey paper we have recalled the main results related with two lines of research on the expressiveness of Linda-like coordination models and on the theory of behavioural contracts. In the third part of the paper, we have discussed how the techniques developed for Linda-like coordination have recently contributed to the advancement of the research in the context of session types for asynchronous communication. Session types, which can be seen as a simplification of contracts, already had a wide application on concurrent programming languages, as, e.g., Haskell [34], Go [39] and Rust [30].

We expect two possible lines for future work: on the one hand, analyse the impact on the theory of contracts of our results for session types (in fact, very few results are present in the literature about contracts for asynchronous communication) and, on the other hand, continue in the context of session types by investigating novel techniques for sound algorithmic characterizations of asynchronous session subtyping.

References

1. F. Barbanera and U. de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *Proc. of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP'10*, pages 155–164. ACM, 2010.
2. M. Boreale and M. Bravetti. Advanced mechanisms for service composition, query and discovery. In *Rigorous Software Engineering for Service-Oriented Systems - Results of the SENSORIA Project on Software Engineering for Service-Oriented Computing*, volume 6582 of *Lecture Notes in Computer Science*, pages 282–301. Springer, 2011.
3. D. Brand and P. Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, 1983.
4. M. Bravetti, M. Carbone, and G. Zavattaro. Undecidability of asynchronous session subtyping. *Inf. Comput.*, 256:300–320, 2017.
5. M. Bravetti, M. Carbone, and G. Zavattaro. On the boundary between decidability and undecidability of asynchronous session subtyping. *Theor. Comput. Sci.*, 722:19–51, 2018.
6. M. Bravetti and G. Zavattaro. Contract based multi-party service composition. In *Proc. of Int. Symposium on Fundamentals of Software Engineering, FSEN'07*, volume 4767 of *Lecture Notes in Computer Science*, pages 207–222. Springer, 2007.
7. M. Bravetti and G. Zavattaro. A theory for strong service compliance. In *Proc. of 9th Int. Conference on Coordination Models and Languages, COORDINATION'07*, volume 4467 of *Lecture Notes in Computer Science*, pages 96–112. Springer, 2007.
8. M. Bravetti and G. Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In *Proc. of 6th Int. Symposium Software Composition, SC'07*, volume 4829 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2007.
9. M. Bravetti and G. Zavattaro. A Foundational Theory of Contracts for Multi-party Service Composition. *Fundamenta Informaticae*, 89(4):451–478, 2008.
10. M. Bravetti and G. Zavattaro. Contract compliance and choreography conformance in the presence of message queues. In *Proc. of 5th Int. Workshop on Web Services and Formal Methods, WS-FM'08*, volume 5387 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2008.
11. M. Bravetti and G. Zavattaro. Choreographies and behavioural contracts on the way to dynamic updates. In *Proc. 1st Workshop on Logics and Model-checking for Self-* Systems, MOD*'14*, volume 168 of *EPTCS*, pages 12–31, 2014.
12. A. Brogi and J. Jacquet. Modeling coordination via asynchronous communication. In *Proc. of 2nd Int. Conference on Coordination Languages and Models, COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 1997.
13. N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing equivalence of Linda coordination primitives. In *Proc. of 4th Workshop on Expressiveness in Concurrency, EXPRESS'97*, volume 7 of *Electr. Notes Theor. Comput. Sci.* Elsevier, 1997.
14. N. Busi, R. Gorrieri, and G. Zavattaro. Three semantics of the output operation for generative communication. In *Proc. of 2nd Int. Conference on Coordination Languages and Models, COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 1997.
15. N. Busi, R. Gorrieri, and G. Zavattaro. Temporary data in shared dataspace coordination languages. In *Proc. of 4th Int. Conference on Foundations of Software*

Science and Computation Structures, FOSSACS'01, volume 2030 of *Lecture Notes in Computer Science*, pages 121–136. Springer, 2001.
16. N. Busi and G. Zavattaro. On the expressiveness of event notification in data-driven coordination languages. In *Proc. of 9th European Symposium on Programming, ESOP'00*, volume 1782 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2000.
17. S. Carpineti, G. Castagna, C. Laneve, and L. Padovani. A formal account of contracts for web services. In *Proc. of 3rd Int. Workshop on Web Services and Formal Methods, WS-FM'06*, volume 4184 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2006.
18. N. Carriero and D. Gelernter. The s/net's linda kernel (extended abstract). In *Proc. of 10th ACM Symposium on Operating System Principles, SOSP'85*, page 160. ACM, 1985.
19. G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for web services. In *Proc. of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'08*, pages 261–272. ACM, 2008.
20. T. Chen, M. Dezani-Ciancaglini, A. Scalas, and N. Yoshida. On the preciseness of subtyping in session types. *Logical Methods in Computer Science*, 13(2), 2017.
21. R. De Nicola, G. L. Ferrari, and R. Pugliese. Coordinating mobile agents via blackboards and access rights. In *Proc. of 2nd Int. Conference on Coordination Languages and Models, COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*, pages 220–237. Springer, 1997.
22. R. De Nicola and R. Pugliese. A process algebra based on LINDA. In *Proc. of 1st Int. Conference on Coordination Languages and Models, COORDINATION'96*, volume 1061 of *Lecture Notes in Computer Science*, pages 160–178. Springer, 1996.
23. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of 25th Int. Colloquium on Automata, Languages and Programming, ICALP'98*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 1998.
24. A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1-2):63–92, 2001.
25. C. Fournet, C. A. R. Hoare, S. K. Rajamani, and J. Rehof. Stuck-free conformance. In *Proc. of 16th International Conference on Computer Aided Verification, CAV'04*, volume 3114 of *Lecture Notes in Computer Science*, pages 242–254. Springer, 2004.
26. S. J. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2-3):191–225, 2005.
27. D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
28. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. of 7th European Symposium on Programming, ESOP'98*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.
29. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. of 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'08*, pages 273–284. ACM, 2008.
30. T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen. Session types for Rust. In *Proc. of 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP'15*, pages 13–22, 2015.
31. D. Kozen. *Automata and computability*. Springer, New York, 1997.

32. C. Laneve and L. Padovani. The *Must* preorder revisited. In *Proc. of 18th Int. Conference Concurrency Theory, CONCUR'07*, volume 4703 of *Lecture Notes in Computer Science*, pages 212–225. Springer, 2007.
33. J. Lange and N. Yoshida. On the undecidability of asynchronous session subtyping. In *Proc. of 20th Int. Conference on Foundations of Software Science and Computation Structures, FOSSACS'17*, volume 10203 of *Lecture Notes in Computer Science*, pages 441–457, 2017.
34. S. Lindley and J. G. Morris. Embedding session types in haskell. In *Proc. of 9th International Symposium on Haskell, Haskell'16*, pages 133–145, 2016.
35. R. Milner. *Communication and concurrency*. Prentice Hall, 1989.
36. M. L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.
37. D. Mostrous and N. Yoshida. Session typing and asynchronous subtyping for the higher-order π-calculus. *Inf. Comput.*, 241:227–263, 2015.
38. D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *Proc. of 18th European Symposium on Programming, ESOP'09*, volume 5502 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 2009.
39. N. Ng and N. Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proc. of 25th International Conference on Compiler Construction, CC'16*, pages 174–184, 2016.
40. OASIS. *Web Services Business Process Execution Language Version 2.0 OASIS Standard*. `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf`.
41. A. Rensink and W. Vogler. Fair testing. *Inf. Comput.*, 205(2):125–198, 2007.
42. J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *J. ACM*, 10(2):217–255, 1963.